compact1, compact2, compact3

java.util

Interface Collection<E>

Type Parameters:

`E` – the type of elements in this collection

All Superinterfaces:

`Iterable<E>`

All Known Subinterfaces:

`BeanContext`, `BeanContextServices`, `BlockingDeque<E>`, `BlockingQueue<E>`, `Deque<E>`, `List<E>`, `NavigableSet<E>`, `Queue<E>`, `Set<E>`, `SortedSet<E>`, `TransferQueue<E>`

All Known Implementing Classes:

`AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, `ArrayBlockingQueue`, `ArrayDeque`, `ArrayList`, `AttributeList`, `BeanContextServicesSupport`, `BeanContextSupport`, `ConcurrentHashMap.KeySetView`, `ConcurrentLinkedDeque`, `ConcurrentLinkedQueue`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, `DelayQueue`, `EnumSet`, `HashSet`, `JobStateReasons`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedHashSet`, `LinkedList`, `LinkedTransferQueue`, `PriorityBlockingQueue`, `PriorityQueue`, `RoleList`, `RoleUnresolvedList`, `Stack`, `SynchronousQueue`, `TreeSet`, `Vector`

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

All general-purpose `Collection` implementation classes (which typically implement `Collection` indirectly through one of its subinterfaces) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type `Collection`, which creates a new collection with the same elements as its argument. In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. There is no way to enforce this convention (as interfaces cannot contain constructors) but all of the general-purpose `Collection` implementations in the Java platform libraries comply.

The "destructive" methods contained in this interface, that is, the methods that modify the collection on which they operate, are specified to throw `UnsupportedOperationException` if this collection does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the collection. For example, invoking the `addAll(Collection)` method on an

unmodifiable collection may, but is not required to, throw the exception if the collection to be added is empty.

Some collection implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the collection may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

It is up to each collection to determine its own synchronization policy. In the absence of a stronger guarantee by the implementation, undefined behavior may result from the invocation of any method on a collection that is being mutated by another thread; this includes direct invocations, passing the collection to a method that might perform invocations, and using an existing iterator to examine the collection.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `contains(Object o)` method says: "returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`." This specification should *not* be construed to imply that invoking `Collection.contains` with a non-null argument `o` will cause `o.equals(e)` to be invoked for any element `e`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two elements. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Implementation Requirements:

`The default method implementations (inherited or otherwise) do not apply any synchronization protocol. If a Collection implementation has a specific synchronization protocol, then it must override default implementations to apply that protocol.`

Since:

`1.2`

See Also:

`Set`, `List`, `Map`, `SortedSet`, `SortedMap`, `HashSet`, `TreeSet`, `ArrayList`, `LinkedList`, `Vector`, `Collections`, `Arrays`, `AbstractCollection`

Method Summary

All Methods **Instance Methods** **Abstract Methods** **Default Methods**

Modifier and Type	Method and Description
`boolean`	**`add`**`(E e)` Ensures that this collection contains the specified element (optional operation).

boolean	**addAll(Collection<? extends E> c)** Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear()** Removes all of the elements from this collection (optional operation).
boolean	**contains(Object o)** Returns `true` if this collection contains the specified element.
boolean	**containsAll(Collection<?> c)** Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals(Object o)** Compares the specified object with this collection for equality.
int	**hashCode()** Returns the hash code value for this collection.
boolean	**isEmpty()** Returns `true` if this collection contains no elements.
Iterator<E>	**iterator()** Returns an iterator over the elements in this collection.
default Stream<E>	**parallelStream()** Returns a possibly parallel `Stream` with this collection as its source.
boolean	**remove(Object o)** Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll(Collection<?> c)** Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	**removeIf(Predicate<? super E> filter)** Removes all of the elements of this collection that satisfy the given predicate.
boolean	**retainAll(Collection<?> c)** Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size()** Returns the number of elements in this collection.
default Spliterator<E>	**spliterator()** Creates a **Spliterator** over the elements in this collection.
default Stream<E>	**stream()** Returns a sequential `Stream` with this collection as its source.
Object[]	**toArray()** Returns an array containing all of the elements in this collection.

<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

size

```
int size()
```

Returns the number of elements in this collection. If this collection contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Returns:

the number of elements in this collection

isEmpty

```
boolean isEmpty()
```

Returns `true` if this collection contains no elements.

Returns:

true if this collection contains no elements

contains

```
boolean contains(Object o)
```

Returns `true` if this collection contains the specified element. More formally, returns `true` if and only if this collection contains at least one element `e` such that `(o==null ? e==null : o.equals(e))`.

Parameters:

o – element whose presence in this collection is to be tested

Returns:

true if this collection contains the specified element

Throws:

ClassCastException – if the type of the specified element is incompatible with this collection (optional)

NullPointerException – if the specified element is null and this collection does not permit null elements (optional)

iterator

`Iterator<E> iterator()`

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).

Specified by:

`iterator` in interface `Iterable<E>`

Returns:

an Iterator over the elements in this collection

toArray

`Object[] toArray()`

Returns an array containing all of the elements in this collection. If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The returned array will be "safe" in that no references to it are maintained by this collection. (In other words, this method must allocate a new array even if this collection is backed by an array). The caller is thus free to modify the returned array.

This method acts as bridge between array-based and collection-based APIs.

Returns:

an array containing all of the elements in this collection

toArray

`<T> T[] toArray(T[] a)`

Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array. If the collection fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this collection.

If this collection fits in the specified array with room to spare (i.e., the array has more elements than this collection), the element in the array immediately following the end of the collection is set to `null`. (This is useful in determining the length of this collection *only* if the caller knows that this collection does not contain any `null` elements.)

If this collection makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

Like the `toArray()` method, this method acts as bridge between array-based and collection-based APIs. Further, this method allows precise control over the runtime type of the output array, and may, under certain circumstances, be used to save allocation costs.

Suppose `x` is a collection known to contain only strings. The following code can be used to dump the collection into a newly allocated array of `String`:

```
String[] y = x.toArray(new String[0]);
```

Note that `toArray(new Object[0])` is identical in function to `toArray()`.

Type Parameters:

T – the runtime type of the array to contain the collection

Parameters:

a – the array into which the elements of this collection are to be stored, if it is big enough; otherwise, a new array of the same runtime type is allocated for this purpose.

Returns:

an array containing all of the elements in this collection

Throws:

ArrayStoreException – if the runtime type of the specified array is not a supertype of the runtime type of every element in this collection

NullPointerException – if the specified array is null

add

```
boolean add(E e)
```

Ensures that this collection contains the specified element (optional operation). Returns `true` if this collection changed as a result of the call. (Returns `false` if this collection does not permit duplicates and already contains the specified element.)

Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add `null` elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.

If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning `false`). This preserves the invariant that a collection always contains the specified element after this call returns.

Parameters:

e – element whose presence in this collection is to be ensured

Returns:

true if this collection changed as a result of the call

Throws:

UnsupportedOperationException – if the add operation is not supported by this collection

ClassCastException – if the class of the specified element prevents it from being added to this collection

NullPointerException – if the specified element is null and this collection does not permit null elements

IllegalArgumentException – if some property of the element prevents it from being added to this collection

IllegalStateException – if the element cannot be added at this time due to insertion restrictions

remove

```
boolean remove(Object o)
```

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element `e` such that `(o==null ? e==null : o.equals(e))`, if this collection contains one or more such elements. Returns `true` if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).

Parameters:

`o – element to be removed from this collection, if present`

Returns:

`true if an element was removed as a result of this call`

Throws:

`ClassCastException – if the type of the specified element is incompatible with this collection (optional)`

`NullPointerException – if the specified element is null and this collection does not permit null elements (optional)`

`UnsupportedOperationException – if the remove operation is not supported by this collection`

containsAll

```
boolean containsAll(Collection<?> c)
```

Returns `true` if this collection contains all of the elements in the specified collection.

Parameters:

`c – collection to be checked for containment in this collection`

Returns:

`true if this collection contains all of the elements in the specified collection`

Throws:

`ClassCastException – if the types of one or more elements in the specified collection are incompatible with this collection (optional)`

`NullPointerException – if the specified collection contains one or more null elements and this collection does not permit null elements (optional), or if the specified collection is null.`

See Also:

`contains(Object)`

addAll

```
boolean addAll(Collection<? extends E> c)
```

Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)

Parameters:

`c – collection containing elements to be added to this collection`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException` – `if the addAll operation is not supported by this collection`

`ClassCastException` – `if the class of an element of the specified collection prevents it from being added to this collection`

`NullPointerException` – `if the specified collection contains a null element and this collection does not permit null elements, or if the specified collection is null`

`IllegalArgumentException` – `if some property of an element of the specified collection prevents it from being added to this collection`

`IllegalStateException` – `if not all the elements can be added at this time due to insertion restrictions`

See Also:

`add(Object)`

removeAll

```
boolean removeAll(Collection<?> c)
```

Removes all of this collection's elements that are also contained in the specified collection (optional operation). After this call returns, this collection will contain no elements in common with the specified collection.

Parameters:

`c – collection containing elements to be removed from this collection`

Returns:

`true if this collection changed as a result of the call`

Throws:

`UnsupportedOperationException` – `if the removeAll method is not supported by this collection`

`ClassCastException` – `if the types of one or more elements in this collection are incompatible with the specified collection (optional)`

`NullPointerException` – `if this collection contains one or more null elements and the specified collection does not support null elements (optional), or if the specified collection is null`

See Also:

`remove(Object), contains(Object)`

removeIf

```
default boolean removeIf(Predicate<? super E> filter)
```

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.

Implementation Requirements:

The default implementation traverses all elements of the collection using its iterator(). Each matching element is removed using Iterator.remove(). If the collection's iterator does not support removal then an UnsupportedOperationException will be thrown on the first matching element.

Parameters:

filter – a predicate which returns true for elements to be removed

Returns:

true if any elements were removed

Throws:

NullPointerException – if the specified filter is null

UnsupportedOperationException – if elements cannot be removed from this collection. Implementations may throw this exception if a matching element cannot be removed or if, in general, removal is not supported.

Since:

1.8

retainAll

```
boolean retainAll(Collection<?> c)
```

Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection.

Parameters:

c – collection containing elements to be retained in this collection

Returns:

true if this collection changed as a result of the call

Throws:

UnsupportedOperationException – if the retainAll operation is not supported by this collection

ClassCastException – if the types of one or more elements in this collection are incompatible with the specified collection (optional)

NullPointerException – if this collection contains one or more null elements and the specified collection does not permit null elements (optional), or if the specified collection is null

See Also:

remove(Object), contains(Object)

clear

```
void clear()
```

Removes all of the elements from this collection (optional operation). The collection will be empty after this method returns.

Throws:

`UnsupportedOperationException` - if the clear operation is not supported by this collection

equals

```
boolean equals(Object o)
```

Compares the specified object with this collection for equality.

While the `Collection` interface adds no stipulations to the general contract for the `Object.equals`, programmers who implement the `Collection` interface "directly" (in other words, create a class that is a `Collection` but is not a `Set` or a `List`) must exercise care if they choose to override the `Object.equals`. It is not necessary to do so, and the simplest course of action is to rely on `Object`'s implementation, but the implementor may wish to implement a "value comparison" in place of the default "reference comparison." (The `List` and `Set` interfaces mandate such value comparisons.)

The general contract for the `Object.equals` method states that equals must be symmetric (in other words, `a.equals(b)` if and only if `b.equals(a)`). The contracts for `List.equals` and `Set.equals` state that lists are only equal to other lists, and sets to other sets. Thus, a custom `equals` method for a collection class that implements neither the `List` nor `Set` interface must return `false` when this collection is compared to any list or set. (By the same logic, it is not possible to write a class that correctly implements both the `Set` and `List` interfaces.)

Overrides:

`equals` in class `Object`

Parameters:

`o` - object to be compared for equality with this collection

Returns:

`true` if the specified object is equal to this collection

See Also:

`Object.equals(Object)`, `Set.equals(Object)`, `List.equals(Object)`

hashCode

```
int hashCode()
```

Returns the hash code value for this collection. While the `Collection` interface adds no stipulations to the general contract for the `Object.hashCode` method, programmers should take note that any class that overrides the `Object.equals` method must also override the `Object.hashCode` method in order to satisfy the general contract for the `Object.hashCode` method. In particular, `c1.equals(c2)` implies that `c1.hashCode()==c2.hashCode()`.

Overrides:

`hashCode` in class `Object`

Returns:

the hash code value for this collection

See Also:

`Object.hashCode()`, `Object.equals(Object)`

spliterator

`default Spliterator<E> spliterator()`

Creates a `Spliterator` over the elements in this collection. Implementations should document characteristic values reported by the spliterator. Such characteristic values are not required to be reported if the spliterator reports `Spliterator.SIZED` and this collection contains no elements.

The default implementation should be overridden by subclasses that can return a more efficient spliterator. In order to preserve expected laziness behavior for the `stream()` and `parallelStream()`} methods, spliterators should either have the characteristic of `IMMUTABLE` or `CONCURRENT`, or be *late-binding*. If none of these is practical, the overriding class should describe the spliterator's documented policy of binding and structural interference, and should override the `stream()` and `parallelStream()` methods to create streams using a `Supplier` of the spliterator, as in:

```
Stream<E> s = StreamSupport.stream(() -> spliterator(), spliteratorCharacteristics)
```

These requirements ensure that streams produced by the `stream()` and `parallelStream()` methods will reflect the contents of the collection as of initiation of the terminal stream operation.

Specified by:

`spliterator` in interface `Iterable<E>`

Implementation Requirements:

The default implementation creates a *late-binding* spliterator from the collections's Iterator. The spliterator inherits the *fail-fast* properties of the collection's iterator.

The created Spliterator reports `Spliterator.SIZED`.

Implementation Note:

The created Spliterator additionally reports `Spliterator.SUBSIZED`.

If a spliterator covers no elements then the reporting of additional characteristic values, beyond that of SIZED and SUBSIZED, does not aid clients to control, specialize or simplify computation. However, this does enable shared use of an immutable and empty spliterator instance (see `Spliterators.emptySpliterator()`) for empty collections, and enables clients to determine if such a spliterator covers no elements.

Returns:

a Spliterator over the elements in this collection

Since:

1.8

stream

`default Stream<E> stream()`

Returns a sequential `Stream` with this collection as its source.

This method should be overridden when the `spliterator()` method cannot return a spliterator that is

IMMUTABLE, CONCURRENT, or *late-binding*. (See spliterator() for details.)

Implementation Requirements:

The default implementation creates a sequential Stream from the collection's Spliterator.

Returns:

a sequential Stream over the elements in this collection

Since:

1.8

parallelStream

```
default Stream<E> parallelStream()
```

Returns a possibly parallel Stream with this collection as its source. It is allowable for this method to return a sequential stream.

This method should be overridden when the spliterator() method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See spliterator() for details.)

Implementation Requirements:

The default implementation creates a parallel Stream from the collection's Spliterator.

Returns:

a possibly parallel Stream over the elements in this collection

Since:

1.8